Report by: Jason Ho JiaSheng (22360143)

1. **public int floodFillCount(int[][] image, int row, int col)**

   The algorithm I chose to use for this question is the breath-first search graph traversal method (BFS). I created 2 queues one to store the row coordinates and the other to store the column coordinates. Also, I used a boolean 2D array - seen, to ensure that each pixel is not visited again. I first will add the starting pixel (row & column) to the respective queues, set its value in the 2D boolean array to true (to prevent visitation twice) then enter a while loop where the pixel removed from the queue's head is checked. If the starting pixel is black/0, then return counter =0 because it is already black.

   The "neighbors" (right,left,bottom & top) of that pixel removed from the queue's head are checked based on if they have the same color (same brightness value as the starting pixel that is not 0) and if they are already visited. If they have the same colour and are not visited yet, that neighbour whether it is left, right, top or bottom will turn to black (get a value of 0) and the counter for contiguous colors will increase. At the same time their value in the boolean seen array would be updated to true(to prevent visitation again). Afterwards that neighbour (row and column coordinates) will be added to the queues and the whole while loop/ checking process is done again. This will stop when there are no more neighboring pixels that have the same initial brightness value/color as the starting pixel and the queues are empty. My code will then return the counter for contiguous colors which represents the total number of times contiguous pixels changed to black. Thus, giving rise to the correct answer.

   The time complexity of my implementation is O(P). This is because in the worst case scenario where for example if like I got an image array like this:

```
 5  5  5
5  5  5
5  5  5
```

If the image 2D array contains all the same values like that, my code implementation will then have to go through **every pixel** to change it to black/ 0 (no matter where the starting pixel is) due to contiguous pixels having the same brightness value. Additionally, I also made sure that no pixel is visited twice(>=2). Furthermore, all of my operations like adding and removing from the queue are O(1), constant time.

2. **public int brightestSquare(int[][] image, int k)**

The algorithm I used for this question is some sort of the Kudane's algorithm. I first create a window in the image array in a rectangle shape with length=1 and breath=k. Then I will sum up the values in that rectangle. This 1 x k rectangle will be formed on every possible element in the downwards direction (there will be overlapping elements & no duplicate rectangles). These new sum values are then put into another 2D array - stripsum. Then I will create another window like a rectangle with length = k and breath=1 and sum up the values. This k x 1 rectangle will be formed on every possible element in the stripsum array in the rightward direction (there will be overlapping elements & no duplicate rectangles). This is equivalent to finding the sum in a k x k square. I also added a comparison operator to compare each k x k square formed to get the maximum sum. This is done in the stripsum array. My code will then return the biggest sum of the k x k square by returning the biggest value in stripsum. For example, let's take a look at an example where the input k =2.

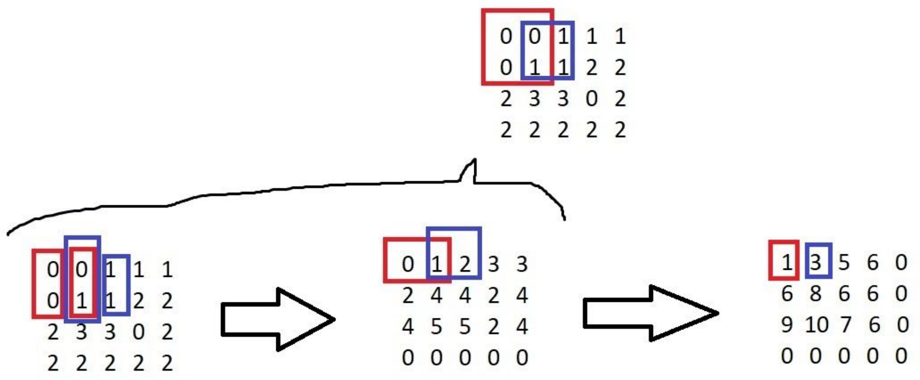
Note: Not all squares/rectangles are drawn.

1. Find the sum of each 1 x 2 rectangle and transfer sum values to stripsum
2. Find the sum of each 2 x 1 rectangle in stripsum and update values.
3. Compare each value in the stripsum array

   Process 1 + 2 would be the same as finding the sum of each 2 x 2 square.

In this case, for example, if you want to find the sum in this 2x2 red square as shown below, we first sum up the columns as shown below (Please look at the red 1 x 2 rectangles at Process 1). Then, we sum up the row(2 x 1 red rectangle at Process 2). We would get a sum value of 1 which is the same result as the sum of the red 2x 2 square.

Off-track: There can be an overlap of elements(look at blue square and rectangles) because we want all possible 2x2 squares that can be formed in this image 2D array.



Then we compare each value the sum in 2x2 square gives in stripsum (in the 3rd process). The maximum value that would be returned would be 10 which is the desired result.

The time complexity of my implementation is O(P). This is because in the worst case scenario where k=1, the square would just include one pixel because it's a 1 x 1 square. We would then have to check **all** the pixels, **one by one** to find the maximum value.

3. **public int darkestPath(int[][] image, int ur, int uc, int vr, int vc)**

The algorithm I used for this question is some sort of a **Dijkstra's algorithm.** More specifically after much research, I found that this problem is basically a minimax path problem where it seeks for a path that minimizes the maximum weight of any of the edges which is similar to our problem which is to find the minimum value out of the maximum values encountered along the paths (many possibilities) from ur, uc to vr, vc.

Since we want to minimize the maximum value of the path, I used a min-heap priority queue to ensure that the smaller brightness value would always be given a higher priority. I also created a 2D array - width, to keep track of the minimum maximum value each pixel comes across. Starting from the source pixel which is added to the priority queue, I will check all its neighbours (right,left, top and bottom) after removal in the while loop with this relaxation:

```
temp = min(width[v], max(width[u], width_between(u, v))) ;
 if(temp < width[v]){
….
}
```

The relaxation would always ensure that I would always get the minimum maximum value as the pixel moves from pixel u to pixel v. If the maximum value found for a pixel in max(width[u], width_between(u, v)) (possibly from different paths) is smaller than its initial value in the width 2D array (if(temp < width[v])), then we would update the initial value in the width[v] to have that value that is smaller (width[v]=temp) and add it back into the priority queue. I also used a 2D boolean array for each time we remove the pixel from the priority queue to check its neighbors to ensure that the program will not revisit past pixels (will not create a loop/cycle). After checking the neighbours of the starting pixel and adding them(if valid) into the priority queue, the while loop starts again with the one with the lowest brightness/value removed first. After going through all the pixels in the image 2D array and updating the width array, my code will return the minimum maximum value the pixel at vr,vc encountered in the width array. Thus, giving rise to the correct answer.

The time complexity of my code implementation is O(Plog(P)). This is because the main loop is executed P times, and each extractmin operation takes O(log(P)) due to using heapify sort in the priority queue. This is because the heapify approach is used to remove the head/root node of the tree. Heapify is based on a binary tree with n elements in it and has a height of log(N), and hence heapify will perform at most log(N) exchanges. This will yield a total time of O(Plog(P)).

During all P operations of the main loop we examine the “neighbours” of the pixel (left,right,top,bottom) exactly once. Because it is examining the left, right, top and bottom of the pixel, each pixel will be examined at a maximum of 4 times. Therefore, we make 4P calls, each of which may cause a change to be performed due to the relaxation:

```
 temp = min(width[v], max(width[u], width_between(u, v))) ;
 if(temp < width[v]){
….
}
```

After each change, it will affect the heap. By adding elements to the priority queue, in the worst case log(P) exchanges will have to be done to sort out the priority of the elements correctly. Therefore, O(log(P)).

Hence, at most O(4Plog(P)).

Therefore, O(Plog(P) + 4Plog(P)) = O(5Plog(P)). And then we simplify it by taking out the constants resulting in O(Plog(P)).

4. <u>public int[][] brightestPixelsInRowSegments(int[][] image, int[][] queries)</u>

I created a **segment tree** and used its operations to solve this problem. I used an array to represent the segment trees. Each node/element in the array is the maximum of all its children. For each element at index i, the left child’s index would be 2i+1 whereas the right child’s index will be 2i+2. To find the index of the parent, is (i-1)/2.

I created a segment tree in each row of the image 2D array. In each segment/row, I will divide the current segment into 2 halves and then call the same procedure onto both halves and for each segment, I stored the maximum value in the segment tree node/array. Each tree would be a complete binary tree because I am dividing each segment into 2 halves at each level.

To query on a range given by the queries like from index 1 to index 3 etc, I have to check 3 conditions which are: Check if range represented by the node/element is completely overlapped/inside by/the given range. Check if the range represented by the node is totally not inside the given range. Lastly, check if the range represented by the node is partially overlapped/inside the given range.

If the segment of the current node is completely part of the given range, then return the maximum value found in the segment. If it is completely outside the given range, then return -1.

If it is partially overlapped, it will be halved again by calling 2 recursion functions, one for start to middle and middle+1 to end. This process will be done for each query. The maximum value found will be assigned to an array storing the maximum values found for each query. After each and every query is processed, this array would be returned. Thus, giving rise to the correct answer.

The time complexity of my code implementation is O(Qlog(C) + P).

Firstly, we have to make each row of the image 2D array into a segment tree. There are R rows in total.

The space complexity to create a suitable array for the segment tree must be O(4C) from an array of size C. This is because we want to display the segment tree in array format. Each node will correspond to some index in the array. In the segment tree, node with index i will have the left child as index= 2i+1 and the right child as index= 2i+2. The relationship between parent and child from the segment tree must be maintained in the array. If the size of the array is power of 2, then we have exactly array_size -1 internal nodes, summing up to 2(array_size)-1 nodes in total. However, if array size is not a power of 2, we will then need the smallest power of 2 which is greater than array size. Therefore, size of memory of array will be 2*(2^ceil(logC)) – 1 or 2*(next power of 2 >= C). This is approximately O(4C).

Because of the space complexity, each operation of constructing the segment tree takes O(C) time. Each node/point in the segment tree is only visited once and the total number of nodes in the tree is 2C-1. Recurrence for the construction is: T(n) = 2T(n/2) + O(1) = O(n). Therefore, construction for each segment tree takes O(C). Overall, the total time complexity in constructing all the segment trees is O(R*C) = O(P).

Secondly, to do each query, we must first consider the level/height of each segment tree. Height of the segment tree will be log(C) because the total number of nodes/vertices in each tree is 2C-1 and because it is a binary tree. There are log c levels in the worst case. There are at most 4 nodes that can be visited at any arbitrary level. The 4 nodes are 2 on the left and 2 on the right. Since the height of the tree is log(C) and at any level only 4 nodes can be visited, it is O(4log(C)). By simplifying it, we will get O(log(C)) time for each querying. Additionally, there are Q times of querying. So overall, O(Qlog(C)) for the querying.

Therefore, the overall time complexity of my code implementation is O(Qlog(C) + P).